I.E.E. 1974 Computer Systems and Technology Conference, London
October 1974

INTEGRATION OF PROTECTION AND PROCEDURES IN A HIGH-LEVEL MINICOMPUTER

B.R. Gaines, Department of Electrical Engineering Science, University of Essex, Colchester, U.K.

M. Haynes and D. Hill, Micro Computer Systems Ltd., Boundary Road, Woking, Surrey, U.K.

Abstract

This paper discusses those aspects of the design of a "high-level" minicomputer specifically orientated to ease and security of software development with high-level languages in a real-time environment.
In particular an implementation of procedure calls is described which both supports the requirements of the common languages and integrates naturally with a ring-structured protection mechanism based on storage segmentation.

INTEGRATION OF PROTECTION AND PROCEDURES IN A HIGH-LEVEL MINICOMPUTER

B.R. Gaines, M. Haynes and D. Hill

1. Introduction

In previous papers[1,2] we have analysed the major current objectives of commercial minicomputer design and have proposed a machine architecture to satisfy them. The design objectives discussed were: (1) Low cost; (2) High reliability; (3) Rapid response; (4) Ease of interfacing; (5) Modularity of configuration; (6) Flexible storage relocation/protection linked to procedure calls; (7) Decoupling peripheral device requirements from the user environment; (8) Consistency and uniformity of treatment of all facilities; (9) A wide range of operand lengths and types; (10) More explicit data structures; (11) Separation of instruction set and order-code; (12) Extensibility through trapping; (13) Dynamic microprogramming; (14) No penalty when the more advanced features are not used.

The first five objectives may be seen as those of the classical minicomputer market. The later objectives reflect the current trend to exploit the availability of low-cost integrated circuit hardware to aid software development, reliability and maintenance. In this paper we shall concentrate on those aspects of our design primarily concerned with simple, secure and swift software engineering, and, in particular, with the integration of protection and procedure calls.

2. Synopsis of Machine Structure

Figure 1 shows the overall structure of the machine. The basic configuration is essentially dual-processor: a simple minicomputer handles all input-output and its low-level processing (buffering, character recognition and conversion, directory lookup, etc.); a 'language processor' executes the main system and user processes, supporting a wide range of operations on operands of various types and lengths together with memory segmentation. The two processors communicate through the store and an interrupt line. The use of I/O processors is common on large machines, and its particular advantages in the present context are discussed in Ref.1. As far as the operating system is concerned it means that the language processor sees a device-independent I/O system, and that the only interrupts to it are significant events requiring major activities such as scheduling.

The 'language processor' is that most relevant to this paper. It consists of three independent units whose functions are:-

2.1 Data operations

Operands are variable in length in units of 8-bits (one byte) from 1 to 16 bytes. There is a single variable-length accumulator (ACC) and virtually all data operations are between an operand in ACC and one in main store with the result in ACC. As well as the 16 operand lengths, there are 16 operand types, including LOGICAL, INTEGER, REAL, COMPLEX, stack MARKER,

B.R. Gaines is in the Department of Electrical Engineering Science, University of Essex.

M. Haynes and D. Hill are with Micro Computer Systems Ltd, Woking.

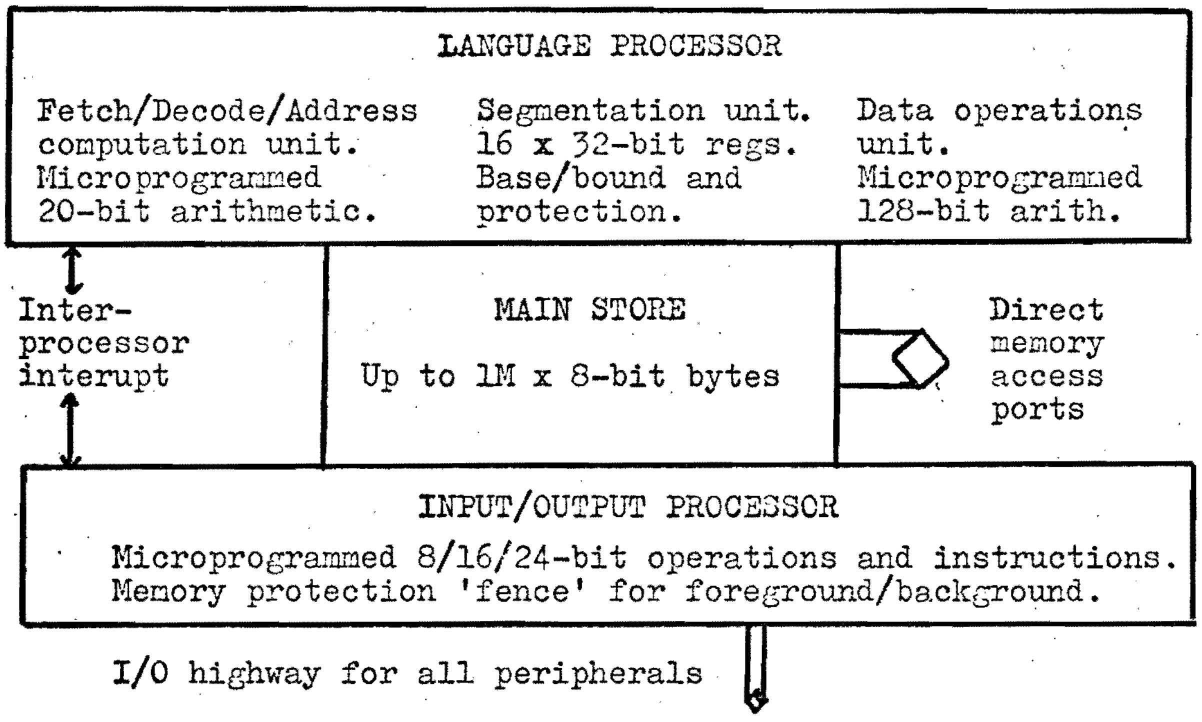
LANGUAGE PROCESSOR
Fetch/Decode/Address computation unit. Microprogrammed 20-bit arithmetic.
Segmentation unit. 16 x 32-bit regs. Base/bound and protection.
Data operations unit. Microprogrammed 128-bit arith.
Inter-processor interupt
MAIN STORE
Up to 1M x 8-bit bytes
Direct memory access ports
INPUT/OUTPUT PROCESSOR
Microprogrammed 8/16/24-bit operations and instructions.
Memory protection 'fence' for foreground/background.
I/O highway for all peripherals

FIGURE 1 Organisation of High-level Minicomputer

REFERENCE, and procedure CALL. The data operations unit provides 128 possible operations on pairs of operands (mixed in length/type) of which 64 operations are hardware defined and the remainder are extracodes. The operations include moves, arithmetic (ADD, SUBTRACT, REVERSE SUBTRACT, MULTIPLY, DIVIDE, REVERSE DIVIDE, COMPARE), logic (Boolean), relational (giving a logical result based on a relational test), shifts, field extraction, etc.

The type/length of an operand are specified in an 8-bit 'descriptor', one of which is associated with the accumulator. Descriptor housekeeping is discussed in detail in Refs.1 & 2: that of an operand in the store may be defined - (a) by default in the processor status word - (b) in certain instructions - (c) in certain forms of indirect address (data structure controllers) - (d) as a tag with the data (used mainly with stacks).

2.2 Segmentation

Sixteen physical segment registers (PSR's) provide access to 16 relocatable segments, each up to 64K bytes in length, variable in units of 16 bytes, in a total store of up to 1M bytes. Each segment may be marked as CODE, DATA or PRIVATE at one of 8 levels corresponding to 8 protection rings: CODE - executable from any ring - no write access - read from rings at or within protection level; DATA - not executable - write from rings at or within protection level - read from anywhere; PRIVATE - not executable - write from rings within protection level - read from rings at or within protection level.

Program segments are accessed through a logical segmentation scheme giving up to 128 shareable logical code segments, and up to 112 logical code segments local to a particular job. One of the PSR's identifies the active logical code segment. Other PSR's give access to frequently-used library and run-time system procedures, executive procedures, and data segments.

2.3 Fetch/decode/address computation

Instructions are encoded in 16-bit 'words' or multiples thereof. There are 'short forms' of the most commonly used operations and accessing modes, but these are all contractions of unrestricted 'full forms' and the use of contracted forms may be left to the linking editor. The address modes include: literal - for constants; direct - to all 16 physical segments; and various forms of indirection. The advent of semiconductor main stores has made it attractive to do away with separate index registers and gain the same facilities through address computation on indirection. Facilities include: add an offset to the indirect address (IA); offset is in bytes or is in operands; offset is literal in instruction, or is in ACC, or is sum of both; operand address is IA or IA + offset; replace IA with IA + offset or leave unmodified.

The indirect addresses have themselves been generalised: an escape code in a (normally one word) indirect address indicates that it is a multi-word 'data structure controller' containing an operand type/length descriptor, bounds on a data structure, accessing mode, and trap information on violation.

3. Modularity, Procedures and Protection

The preceding synopsis of the machine indicates how we have attempted to fulfil objectives 7 through 11 of the introduction and provide a mini-computer that avoids the need for hand-coded programs to take advantage of, or overcome, the quirks of the machine - in particular, a minicomputer that can support the full compilation of extended versions of current high-level languages. Objective 6, which is the topic of this section, requires further expansion.

We had in mind that the larger minicomputers are commonly used on-line to other plant, performing a variety of real-time control and data-acquisition tasks in the foreground with some data-processing in the back-ground. As always, software development on such systems does not cease with their installation. It presents major problems because: (a) it is unlikely to be economic to provide a duplicate configuration for develop-ment only; (b) the new software is rarely an unrelated module that can be developed in isolation from the existent system. What happens to these systems in practice is that they 'freeze' at the point where the danger of side-effects in modifying them outweighs the potential advantages of doing so. This balance could be tilted very favourably if new software modules could be added within an environment sufficiently isolated to ensure adequate system operation despite any malfunction in the new module, and accessed through standardised interfaces providing complete error control.

As re-discovered many times in many fields, isolation and standardisation have their own dangers if pursued too far, leading to inefficiency and inflexibility. Enforcement at a hardware level should be the minimum logically necessary, and constraints beyond this should be conditional on requests. It is enabling these requests to be embedded in a simple and natural manner within current languages that represents a major hurdle for the machine designer. Too much detailed effort in setting up different protection structures for separate parts of a software system is self-defeating in that it will not be done, or is itself prone to error. Organick[3] reports how little-used has been the 32-layer ring structure of MULTICS and its associated protection facilities. The binary foreground/ background system has much to commend it in its simplicity and ease of conceptualisation.

We finally selected an 8-level protection system in which a given procedure could execute within any of 8 'rings' and, according to the ring, would have various access rights to data segments, executive routines, input-output, etc. We envisaged a binary tree structure for the use of these rings with 0-3 for system programs, 4-7 for user programs - even numbered rings for normal use, odd-numbered for development - 4/5 for a user-level executive (control and privileged programs) - 6/7 for applications programs. As in MULTICS, the rings are in descending order of capability, and a resource available in an outer (higher-numbered) ring is also available in those of lower number.

The following sub-sections describe how the procedure call mechanism interacts naturally with the protection mechanisms to allow simple and natural control of protection through procedure calls in a high-level language.

3.1 Procedure call and workspace

Figure 2 illustrates the environment for a procedure call. Two of the PSR's have specialised functions in that they provide local workspace to procedures and the segments are physically mapped within other, enclosing segments. To a particular procedure they are physical segments 0 and 1, but the actual store areas they access varies with each call. The STACK FRAME SEGMENT provides dynamic local workspace and is mapped in stacklike fashion within the STACK SEGMENT. At the base of the FRAME is a pointer to the top of the current evaluation stack - this is used implicitly by zero-address instructions (complete 1-address and 0-address families are provided). The OWN FRAME SEGMENT provides static local workspace and is always mapped into the same area on each call of a procedure. The OWN FRAMEs of all procedures within a given code segment have to be mapped somewhere within an OWN SEGMENT corresponding to the code segment.

The FRAME SEGMENT's provide the two types of local workspace required by procedures, e.g. in FORTRAN (FORTREV specification 4) subprogram locals will normally be in the STACK FRAME unless they are SAVEd in which case they will be in the OWN FRAME (and retain their values between calls on the subprogram). The use of segment registers to access local workspace has obvious advantages in allowing short references to be generated giving fast, compact programs. The fact that the protection status of the enclosing segment need not be the same as that of the enclosed FRAME (FRAMEs are always read/write enabled at all levels) is the key to other benefits. For example an untrusted procedure may not have write-access to the whole of the OWN SEGMENT but only to its local area within it. When a similar consideration is applied to the STACK SEGMENT a natural mechanism for outward calls and inward returns is generated.

Procedures are called via a 16-bit two-part transfer vector giving the logical segment number of the procedure and its number within a table of pointers to procedure heads (gates) within that segment. The procedure head specifies the offset and size of its OWN FRAME, the minimum level at which it may be executed, and other status information. The actual minimum ring in which a procedure may execute may be restricted by this head, the level of its segment (to trap loader errors) and by the caller. To simplify parameter validation a procedure may raise its level, e.g. to that of the caller, for part of its execution - it may drop it freely subject to the (unchangeable) minimum.

3.2 Outward calls and inward returns

The STACK SEGMENT has associated with an additional register, the BAR, such that the segment is accessed normally below the BAR (it is normally set up as a DATA segment of the innermost ring), but is read/write enabled above the BAR at all levels. The BAR is not changed for calls on procedures in the same or inner rings, but an outward call causes it to be advanced to the base of the new stack frame, thereby write-disabling previous STACK FRAMEs to the called procedure. All the return information is stored in the STACK FRAME of the caller (the address of the base of this frame is stored at the top of it where it is accessible as the location below the new STACK FRAME), and hence becomes inaccessible on an outward call. Thus inward returns can be made with the assurance that the return information is valid (they also reset the BAR).

If a violation occurs in a procedure for which no error provision has been made, then an error-return is made to the caller indicated by the BAR (the last outward call). Hence proper control can be provided simply and automatically over unforeseen errors in untrusted packages.

3.3 Parameter passing

The use of operand type/length descriptors allows parameters to be passed to procedures with the minimum of constraint on both caller and procedure. Parameters are placed on the evaluation stack (and hence are automatically tagged as to type and length) with the final parameter in the ACC being the number of parameters passed. A pointer to the previous evaluation stack is passed automatically to the caller. The caller is able to unstack reference variables pointing to the parameters and hence is able to use them regardless of type. Hence parameters may be passed not only differing in type and length from that expected, but they may also be passed in different ways without explicit action by the called procedure.

4. Conclusions

This paper has given a brief over-view of some aspects of the design of a 'high-level' minicomputer specifically orientated to ease and security of software development with high-level languages in a real-time environment. It indicates an important trend in minicomputer design made possible by the decline in costs of the basic circuit technology.

5. References

1. Gaines, B.R., Facey, P.V., Williamson, F.K., Maine, J.A.: 'Design Objectives for a Descriptor Organized Minicomputer' European Computing Congress Conference Proceedings, London: Online 1974 29-45.

2. Williamson, F.K., Gaines, B.R., Maine, J.A., Facey, P.V.: 'A High-Level Minicomputer' Proc. IFIP Congress, Stockholm 1974.

3. Organick, E.I.: The Multics System, M.I.T. Press 1972.

4. Working Document of ANSC X3J3 (73-06-09) Bell Labs., Holmdel, N.J., U.S.A.

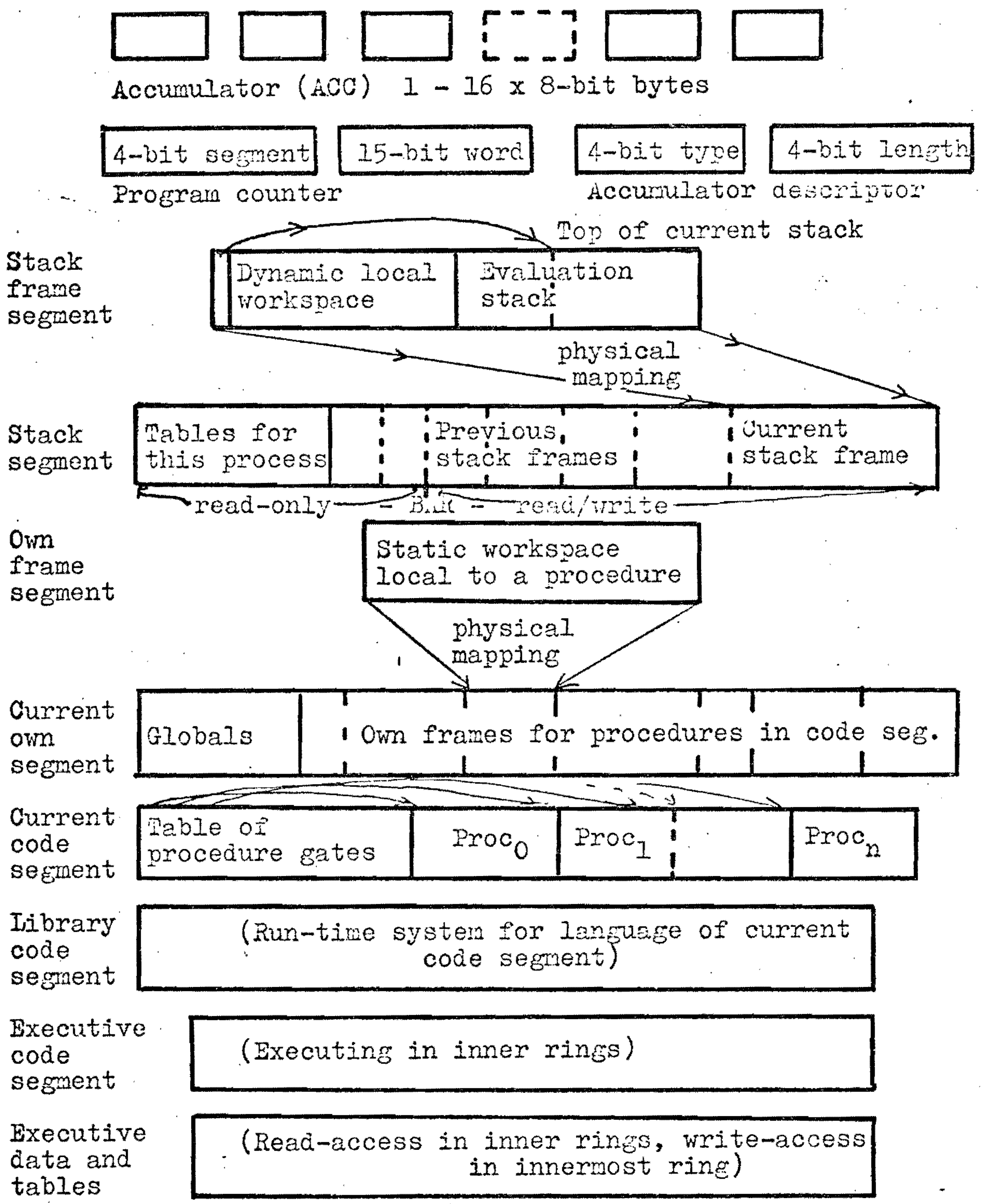

Figure 2 The Environment for a Procedure Call